

# Dictionary Matching with One Gap<sup>\*</sup>

Amihood Amir<sup>1,2,\*\*</sup>, Avivit Levy<sup>3</sup>, Ely Porat<sup>1</sup>, and B. Riva Shalom<sup>3</sup>

<sup>1</sup> Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel.

E-mail: {amir, porately}@cs.biu.ac.il

<sup>2</sup> Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218.

<sup>3</sup> Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel.

Email: {avivitlevy, rivash}@shenkar.ac.il

**Abstract.** The dictionary matching with gaps problem is to preprocess a dictionary  $D$  of  $d$  gapped patterns  $P_1, \dots, P_d$  over alphabet  $\Sigma$ , where each gapped pattern  $P_i$  is a sequence of subpatterns separated by bounded sequences of don't cares. Then, given a query text  $T$  of length  $n$  over alphabet  $\Sigma$ , the goal is to output all locations in  $T$  in which a pattern  $P_i \in D$ ,  $1 \leq i \leq d$ , ends. There is a renewed current interest in the gapped matching problem stemming from cyber security. In this paper we solve the problem where all patterns in the dictionary have one gap with at least  $\alpha$  and at most  $\beta$  don't cares, where  $\alpha$  and  $\beta$  are given parameters. Specifically, we show that the dictionary matching with a single gap problem can be solved in either  $O(d \log d + |D|)$  time and  $O(d \log^\epsilon d + |D|)$  space, and query time  $O(n(\beta - \alpha) \log \log d \log^2 \min\{d, \log |D|\} + occ)$ , where  $occ$  is the number of patterns found, or preprocessing time and space:  $O(d^2 + |D|)$ , and query time  $O(n(\beta - \alpha) + occ)$ , where  $occ$  is the number of patterns found. As far as we know, this is the best solution for this setting of the problem, where many overlaps may exist in the dictionary.

## 1 Introduction

Pattern matching has been historically one of the key areas of computer science. It contributed many important algorithms and data structures that made their way to textbooks, but its strength is that it has been contributing to applied areas, from text searching and web searching, through computational biology and to cyber security. One of the important variants of pattern matching is pattern matching with variable length gaps. The problem is formally defined below.

### Definition 1. Gapped Pattern

A *Gapped Pattern* is a pattern  $P$  of the form  $p_1 \{\alpha_1, \beta_1\} p_2 \{\alpha_2, \beta_2\} \dots \{\alpha_{k-1}, \beta_{k-1}\} p_k$ , where each subpattern  $p_j$ ,  $1 \leq j \leq k$  is a string over alphabet  $\Sigma$ , and

---

<sup>\*</sup> This research was supported by the Kabarnit Cyber consortium funded by the Chief Scientist in the Israeli Ministry of Economy under the Magnet Program.

<sup>\*\*</sup> Partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.

$\{\alpha_j, \beta_j\}$  refers to a sequence of at least  $\alpha_j$  and at most  $\beta_j$  don't cares between the subpatterns  $P_j$  and  $P_{j+1}$ .

**Definition 2.** The Gapped Pattern Matching Problem:

*Input:* A text  $T$  of length  $n$ , and a gapped pattern  $P$  over alphabet  $\Sigma$

*Output:* All locations in  $T$ , where the pattern  $P$  ends.

The problem arose a few decades ago by real needs in computational biology applications [20,12,22,14]. For example, the PROSITE database [14] supports queries for proteins specified by gaps.

The problem has been well researched and many solutions proposed. The first type of solutions [7,19,23] consider the problem as a special case of regular expression. The best time achieved using this method is  $O(n(B|\Sigma| + m))$ , where  $n$  is the text length,  $B = \sum_{i=1}^k \beta_i$  (the sum of the upper bounds of the gaps), and  $m = \sum_{i=1}^k p_i$  (the length of the non-gapped part of the pattern).

Naturally, a direct solution of the gapped pattern matching problem should have a better time complexity. Indeed, such a solution exists [8] whose time is essentially  $O(nk)$ .

A further improvement [18,24,6] analyses the time as a function of *socc*, which is the number of times all segments  $p_i$  of the gapped pattern appear in the text. Clearly  $socc \leq nk$ .

Rahman et al. [24] suggest two algorithms for this problem. In the first, they build an Aho-Corasick [1] pattern matching machine from all the subpatterns and use it to go over the text. Validation of subpatterns appearances with respect to the limits of the gaps is performed using binary search over previous subpatterns locations. Their second algorithm uses a suffix array built over the text to locate occurrences of all subpatterns. For the validation of occurrences of subpattern, they use Van Emde Boas data structure [26] containing ending positions of previous occurrences. In order to report occurrences of the gapped pattern in both algorithms, they build a graph representing legal appearances of consecutive subpatterns. Traversing the graph yields all possible appearances of the pattern.

Their first algorithm works in time  $O(n + m + socc \log(\max_j gap_j))$  where  $m$  is the length of the pattern (not including the gaps), *socc* is the total number of occurrences of the subpatterns in the text and  $gap_j = \beta_j - \alpha_j$ . The time requirements of their second algorithm is  $O(n + m + socc \log \log n)$  where  $n$  is the length of the text,  $m$  is the length of the pattern (not including the gaps) and *socc* is the total number of occurrences of the subpatterns in the text. The DFS traversal on the subpatterns occurrences graph, reporting all the occurrences is done in  $O(k \cdot occ)$  where *occ* is the number of occurrences of the complete pattern  $P$  in the text.

Bille et al. [6] also consider string matching with variable length gaps. They present an algorithm using sorted lists of disjoint intervals, after traversing the text with Aho-Corasick automaton. Their time complexity is  $O(n \log k + m + socc)$  and space  $O(m + A)$ , where  $A$  is the sum of the lower bounds of the lengths of

the gaps in the pattern  $P$  and  $socc$  is the total number of occurrences of the substrings in  $P$  within  $T$ .

Kucherov and Rusinowitch [16] and Zhang et al. [29] solved the problem of matching a set of patterns with variable length of don't cares. They considered the question of determining whether one of the patterns of the set matches the text and report a leftmost occurrence of a pattern if there exists one. The algorithm of [16] has run time of  $O((|t|+|D|)\log|P|)$ , where  $|D|$  is the total length of keywords in every pattern of the dictionary  $D$ . The algorithm of [29] takes  $O((|t|+dk)\log dis/\log\log dis)$  time, where  $dk$  is the total number of keywords in every pattern of  $P$ , and  $dis$  is the number of distinct keywords in  $D$ .

There is a renewed current interest in the gapped matching problem stemming from a crucial modern concern - cyber security. Network intrusion detection systems perform protocol analysis, content searching and content matching, in order to detect harmful software. Such malware may appear on several packets, and thus the need for gapped matching [15]. However, the problem becomes more complex since there is a large list of such gapped patterns that all need to be detected. This list is called a *dictionary*. Dictionary matching has been amply researched in computer science (see e.g. [1,4,3,9,5,2,10]). We are concerned with a new dictionary matching paradigm - *dictionary matching with gaps*. Formally:

**Definition 3.** The Dictionary Matching with gaps (*DMG*) Problem:

*Input:* A text  $T$  of length  $n$  over alphabet  $\Sigma$  and a dictionary  $D$  over alphabet  $\Sigma$  consisting of  $d$  gapped patterns  $P_1, \dots, P_d$ .

*Output:* All locations in  $T$ , where a pattern  $P_i$ , for  $1 \leq i \leq d$  ends.

The *DMG* problem has not been sufficiently studied yet. Haapasalo et al. [13] give an on-line algorithm for the general problem. Their algorithm is based on locating “keywords” of the patterns in the input text, that is, maximal substrings of the patterns that contain only input characters. Matches of prefixes of patterns are collected from the keyword matches, and when a prefix constituting a complete pattern is found, a match is reported. In collecting these partial matches they avoid locating those keyword occurrences that cannot participate in any prefix of a pattern found thus far. Their experiments show that this algorithm scales up well, when the number of patterns increases. They report at most one occurrence for each pattern at each text position. The time required for their algorithm is  $O(n \cdot SUF + occ \cdot PREF)$ , where  $n$  is the size of the text,  $SUF$  is the maximal number of suffixes of a keyword that are also keywords, and  $PREF$  denotes the number of occurrences in the text of pattern prefixes ending with a keyword.

Nevertheless, more research on this problem is needed. First, in many applications it is necessary to report all patterns appearances. Moreover, as far as we know [27], these Aho-Corasick automaton based methods fail when applied to real data security, which contain many overlaps in the dictionary patterns, due to overhead in the computation when run on several ports in parallel. Therefore, other methods should be developed and combined with the existing ones in order to design efficient practical solutions.

**Results.** In this paper, we indeed suggest other directions for solving the problem. We focus on the *DMG* problem where the gapped patterns in the dictionary  $D$  have only a single gap, i.e., we consider the case of  $k = 1$  implying each pattern  $P_i$  consists of two subpatterns  $P_{i,1}, P_{i,2}$ . In addition, we consider the same gaps limits,  $\alpha$  and  $\beta$ , apply to all patterns in  $P_i \in D$ ,  $1 \leq i \leq d$ . We prove:

**Theorem 1.** *The dictionary matching with a single gap problem can be solved in:*

1. *Preprocessing time:*  $O(d \log d + |D|)$ .  
*Space:*  $O(d \log^\varepsilon d + |D|)$ , for arbitrary small  $\varepsilon$ .  
*Query time:*  $O(n(\beta - \alpha) \log \log d \log^2 \min\{d, \log |D|\} + occ)$ , where  $occ$  is the number of patterns found.
2. *Preprocessing time:*  $O(d^2 + |D|)$ .  
*Space:*  $O(d^2 + |D|)$ .  
*Query time:*  $O(n(\beta - \alpha) + occ)$ , where  $occ$  is the number of patterns found.

Note that,  $|D|$  is the sum of lengths of all patterns in the dictionary, *not including* the gaps sizes.

The paper is organized as follows. In Sect. 2 we describe our basic method based on suffix trees and prove the first part of Theorem 1. In Sect. 3 we describe how this algorithm query time can be improved while doing more work in the preprocessing time and prove the second part of Theorem 1. We also discuss efficient implementations of both algorithms using text splitting in Subsect. 3.1. Section 4 concludes the paper and poses some open problems.

## 2 Bidirectional Suffix Trees Algorithm

The basic observation used by our algorithm is that if a gapped pattern  $P_i$  appears in  $T$ , then searching to the left from the start position of the gap we should find the reverse of the prefix  $P_{i,1}$ , and searching to the right from the end position of the gap we should find the suffix  $P_{i,2}$ . A similar observation was used by Amir et al. [5] to solve the dictionary matching with one mismatch problem. Their problem is different from the *DMG* problem, since they consider a single mismatch while in our problem a gap may consist of several symbols. Moreover, a mismatch symbol appears both in the dictionary pattern and in the text, while in the *DMG* problem the gap implies skipping symbols only in the text. Nevertheless, we show that their idea can also be adopted to solve the *DMG* problem.

Amir et al. [5] use two suffix trees: one for the concatenation of the dictionary and the other for the reverse of the concatenation of the dictionary. Combining this with set intersection on tree paths, they solved the dictionary matching with one mismatch problem in time  $O(n \log^{2.5} |D| + occ)$  where  $n$  is the length of the text,  $|D|$  is the sum of the lengths of the dictionary patterns, and  $occ$  is the number of occurrences of patterns in the text. Their preprocessing requires  $O(|D| \log |D|)$ . We use the idea to design an algorithm to the *DMG* problem.

A naive method is to consider matching the prefixes  $P_{i,1}$  for all  $1 \leq i \leq d$ , and then look for the suffixes subpatterns  $P_{i,2}$ ,  $1 \leq i \leq d$ , after the appropriate gap and intersect the occurrences to report the dictionary patterns matchings. However, as some of the patterns may share subpatterns and some subpatterns may include other subpatterns, there may be several distinct subpatterns occurring at the same text location, each of different length. Therefore, we need to search for the suffixes  $P_{i,2}$ ,  $1 \leq i \leq d$ , after several gaps, each beginning at the end of a matched prefix  $P_{i,1}$ ,  $1 \leq i \leq d$ . To avoid multiple searches we search all subpatterns  $P_{i,1}$ ,  $1 \leq i \leq d$  that *end* at a certain location. Note that in order to find all subpatterns ending at a certain location of the text we need to look for them backwards and find their reverse  $P_{i,1}^R$ .

In the preprocessing stage we concatenate the subpatterns  $P_{i,2}$ ,  $1 \leq i \leq d$  of the dictionary separated by the symbol  $\$ \notin \Sigma$  to form a single string  $S$ . We repeat the procedure for the subpatterns  $P_{i,1}$ ,  $1 \leq i \leq d$ , to form a single string  $F$ . We construct a suffix tree  $T_S$  of the string  $S$ , and another suffix tree  $T_{FR}$  of the string  $F^R$ , which is the reverse of the string  $F$ .

We then traverse the text by inserting suffixes of the text to the  $T_S$  suffix tree. When we pass the node of  $T_S$  for which the path from the root is labeled  $P_{i,2}$ , it implies that this subpattern occurs in the text starting from the beginning of the text suffix. We then should find whether  $P_{i,1}$  also appears in the text within the appropriate gap from the occurrences of the  $P_{i,2}$  subpatterns. To this end, we go backward in the text skipping as many locations as the gap requires and inserting the reversed prefix of the text to  $T_{FR}$ . If a node representing  $P_{i,1}$  is encountered, we can output that  $P_i$  appears in the text.

Note that several dictionary subpatterns representative nodes may be encountered while traversing the trees. Therefore, we should report the intersection between the subpatterns found from each traversal. We do that by efficient intersection of labels on tree paths, as done in Amir et al. [5], using range queries on a grid. However, since some patterns may share subpatterns, we do not label the tree nodes by the original subpatterns they represent, as done in [5]. Instead, we mark the nodes representing subpatterns numerically in a certain order, hereafter discussed, regardless to the origin of the subpatterns ending at those nodes.

In order to be able to trace the identity of the patterns from the nodes marking, we keep two arrays  $A_F$  and  $A_S$ , both of size  $d$ . The arrays contain linked lists that identify when subpatterns are shared among several dictionary patterns. These arrays are filled as follows:  $A_F[g] = i$  if  $P_{i,1}^R$  is represented by node labelled  $g$  in  $T_{FR}$  and  $A_S[h] = i$  if  $P_{i,2}$  is represented by the node labelled by  $h$  in  $T_S$ . In addition,  $A_F[g]$  is linked to  $h$  and vice versa, if  $g, h$  represent two subpatterns of the same pattern of the dictionary.

Every pattern  $P_i$  is represented as a point  $i$  on a grid of size  $d \times d$ , denoted by  $\langle g, A_F[g].link \rangle$ , that is, the  $x$ -coordinate is the mark of the node representing  $P_{i,1}^R$  in  $T_{FR}$ , and the  $y$ -coordinate is the mark of the node representing  $P_{i,2}$  in  $T_S$ . Now, if we mark the nodes representing the end of subpatterns so that the marks on a path are consecutive numbers, then the problem of intersection of labels

on tree paths can be reduced to range queries on a grid in the following way. Let the first and last mark on the relevant path in  $T_{FR}$  be  $g, g'$  and, similarly, on the path in  $T_S$  let the first and last mark be  $h, h'$ . Thus, points  $\langle x, y \rangle$  on the grid where  $g \leq x \leq g'$  and  $h \leq y \leq h'$ , represent patterns in the dictionary for which both subpatterns appear at the current check. A range query can be solved using the algorithm of [11].

We use the decomposition of a tree into vertical path for the nodes marking, suggested by [5], though we use it differently. A vertical path is defined as follows.

**Definition 4.** [5] *A vertical path of a tree is a tree path (possibly consisting of a single node) where no two nodes on the path are of the same height.*

After performing the decomposition, we can traverse the vertical paths and mark by consecutive order all tree nodes representing the end of a certain subpattern appearing in  $T_S$  or its reverse appearing in  $T_{FR}$ . Since some subpatterns may be shared by several dictionary patterns, there are *at most*  $d$  marked nodes at each of the suffixes trees.

Note that due to the definition of vertical path there may be several vertical paths that have a non empty intersection with the unique path from the root to a specific node. Hence, when considering the intersection of marked nodes on the path from the root till a certain node marked by  $g$  in  $T_{FR}$  and the marked nodes on the path from the root till a node marked by  $h$  in  $T_S$ , we actually need to check the intersection of all vertical paths that are included in the path from the root to  $g$  with all vertical paths that are included in the path from the root to  $h$ .

The algorithm appears in Figure 1.

**Lemma 1.** *The intersection between the subpatterns appearing at location  $t_\ell$  and the reversed subpatterns ending at  $t_{\ell-gap-1}$  can be computed in time  $O(occ + \log \log d \log^2 \min\{d, \log |D|\})$ , where  $occ$  is the number of patterns found. The preprocessing requires  $O(|D| + d \log d)$  time and  $O(|D| + d \log^\epsilon d)$  space, for arbitrary small  $\epsilon$ .*

At each of the  $O(n)$  relevant locations of the text, the algorithm inserts the current suffix of the text to  $T_S$  using Weiner's algorithm [28]. For each of the prefixes defined by all  $\beta - \alpha + 1$  possible specific gaps we insert its reverse to  $T_{FR}$ . As explained in [5], the navigation on the suffix tree and reverse suffix tree can be done in amortized  $O(1)$  time per character insertion. Note that each character is inserted to  $T_S$  once and to  $T_{FR}$   $O(\beta - \alpha)$  times. This concludes the proof of the first part of Theorem 1.

### 3 Intersection by Lookup Table

If a very fast query time is crucial and we are willing to pay in preprocessing time, we can solve the problem of intersection between the appearances of subpatterns on the paths of  $T_{FR}, T_S$  using a lookup table.

**Preprocessing:**

- 1  $F = P_{1,1} \$ P_{2,1} \$ \dots P_{d,1}$ .
- 2  $S = P_{1,2} \$ P_{2,2} \$ \dots P_{d,2}$ .
- 3  $T_S \leftarrow$  a suffix tree of  $S$ .
- 4  $T_{FR} \leftarrow$  a suffix tree of the  $F^R$ .
- 5 **For** every edge  $(u, v) \in \{T_S, T_{FR}\}$  with label  $y \$ z$ , where  $y, z \in \Sigma^*$
- 6     Break  $(u, v)$  into  $(u, w)$  and  $(w, v)$  labelling  $(u, w)$  with  $y$  and  $(w, v)$  with  $\$ z$ .
- 7     Decompose  $T_{FR}$  into vertical paths.
- 8     Mark the nodes representing  $P_{i,1}$  on the vertical paths of  $T_{FR}$ .
- 9     Decompose  $T_S$  into vertical paths.
- 10    Mark the nodes representing  $P_{i,2}$  on the vertical paths of  $T_S$ .
- 11    Preprocess the points according to the patterns for range queries.

**Query:**

- 12 **For**  $\ell = \min_i \{|P_{i,1}|\} + \alpha$  to  $n$
  - 13     Insert  $t_\ell t_{\ell+1} \dots t_n$  to  $T_S$ .
  - 14      $h \leftarrow$  node in  $T_S$  representing suffix  $t_\ell t_{\ell+1} \dots t_n$ .
  - 15     **For**  $f = \ell - \alpha - 1$  to  $\ell - \beta - 1$
  - 16         Insert  $t_f t_{f+1} \dots t_1$  to  $T_{FR}$ .
  - 17          $g \leftarrow$  node in  $T_{FR}$  representing  $t_f t_{f+1} \dots t_1$ .
  - 18         **For** every vertical path on the the path  $p$  from the root to  $h$
  - 19             **For** every vertical path  $p'$  on the path from the root to  $g$
  - 20                 Perform a range query on a grid with the first and last marks of  $p$  and  $p'$
  - 21                 Report appearance for every  $P_i$  where point  $i$  appears in the specified range.
- 

**Fig. 1.** Dictionary matching with a single gap algorithm, intersection is computed by range queries on a grid.

The *inter* table is of size  $d \times d$ , where  $inter[g, h]$  refers to the set of all indices  $i$  of patterns  $P_i$  such that  $P_{i,1}^R$  appears on the path from the root of  $T_{FR}$  till the node marked by  $g$  and  $P_{i,2}$  appears on the path from the root of  $T_S$  till the node marked by  $h$ . We fill the table using dynamic programming procedure. Consequentially, labelling the nodes representing subpatterns, can be done by any numbering system, guaranteeing that nodes closer to the root are labelled by smaller numbers than nodes farther from the root, such as the BFS order.

Saving at every entry all the relevant pattern indices causes redundancy in case subpatterns include others as their prefix or suffix. In order to save every possible occurrence only once, we save pattern index  $i$  only at entry  $inter[g, h]$  where  $g, h$  are the nodes respectively representing both subpatterns of  $P_i$  in the suffix trees. Note that at most one index can be saved at  $inter[g, h].index$  as two patterns are bound to differ by at least one subpattern. The filling of these  $d$  fields is done in the preprocessing.

We hereafter prove that besides the *index* field, merely 3 links are required for every  $inter[g, h]$  :

1. A link to  $inter[g', h]$  in case node  $h$  represents subpattern  $P_{i,2}$  and  $g'$  is the maximal labelled ancestor of node  $g$ , representing  $P_{i,1}$ . We call this link an *up* link.
2. A link to cell  $inter[g, h']$  in case node  $g$  represents the subpattern  $P_{i,1}$  and  $h'$  is the maximal labelled ancestor of node  $h$  representing  $P_{i,2}$ . We call this link a *left* link.
3. A link to cell  $[prev^*(g), prev^*(h)]$ , where  $prev^*(g)$  and  $prev^*(h)$  are the closest marked ancestors of the nodes marked by  $g$  and  $h$  where  $inter[prev^*(g), prev^*(h)]$  has a pattern index or non null *up* or *left* link.

The recursive rule for constructing the lookup table is described in the following lemma.

**Lemma 2. The Recursive Rule**

Let  $prev(x)$  be the maximal labelled ancestor of the node labelled by  $x$ .

$$\begin{aligned}
 inter[g, h].up &= \begin{cases} [prev(g), h] & \text{if } inter[prev(g), h].index \neq null \\ inter[prev(g), h].up & \text{otherwise} \end{cases} \\
 inter[g, h].left &= \begin{cases} [g, prev(h)] & \text{if } inter[g, prev(h)].index \neq null \\ inter[g, prev(h)].left & \text{otherwise} \end{cases} \\
 inter[g, h].prev &= \begin{cases} [prev(g), prev(h)] & \text{if } inter[prev(g), prev(h)].index \neq null \\ & \text{or } inter[prev(g), prev(h)].up \neq null \\ & \text{or } inter[prev(g), prev(h)].left \neq null \\ inter[prev(g), prev(h)].prev & \text{otherwise} \end{cases}
 \end{aligned}$$

*Proof.* Every  $inter[g, h]$  entry for  $1 \leq g, h \leq d$ , has to contain links to all entries containing indices of patterns whose first subpattern is represented by node  $g$  or its ancestors in  $T_{FR}$  and its second subpattern is represented by node  $h$  or its ancestors in  $T_S$ . Assume, without loss of generality, that the marks on the path from the root of  $T_{FR}$  till the node marked by  $g$  are  $g'_1, g'_2, \dots, g'_a, g$  and the marks on the path from the root of  $T_S$  till the node marked by  $h$  are  $h'_1, h'_2, \dots, h'_b, h$ . There are four cases of relevant entries, for each we prove the correctness of the recursive rule.

1. Case 1: In case the pair of labels  $\langle g, h \rangle$  represent pattern  $P_i$ , then assign  $inter[g, h].index$  by  $i$  in the preprocess. No recursion is required.
2. Case 2: Some  $\{g'_x\}$  represent the reverse of first subpattern of some dictionary patterns (as theses subpatterns include others as their suffixes), and these patterns share the second subpattern where  $h$  represent this second subpattern. In such a case  $inter[g, h]$  should be linked to all entries  $\{inter[g'_x, h]\}$ . Nevertheless, saving a link to the ancestor node with maximal label, is sufficient, since all other relevant patterns can be reached by recursively following *up* links starting from  $inter[g'_x, h]$ . Therefore, we consider  $prev(g)$  as the maximal labeled ancestor and if  $inter[prev(g), h]$  includes an index, we assign *up* with  $[prev(g), h]$ . Otherwise we are seeking the same ancestor  $inter[prev(g), h]$  looked for, for its *up* link, hence we assign *up* with  $inter[prev(g), h].up$ .



3. Case 3: Some  $\{h'_y\}$  represent the second subpattern of some dictionary patterns (as these subpatterns include others as their prefixes), and these patterns share the first subpattern where  $g$  represent the reverse of this first subpattern. Due to similar arguments we assign the *left* link either with  $[g, prev(h)]$  or with  $inter[g, prev(h)].left$ .
4. Case 4: Some ancestors of nodes  $g$  and  $h$  represent both subpatterns of dictionary patterns. Note that it must be ancestors to both nodes as the previous cases dealt with representations of patterns using the nodes  $g$  or  $h$  themselves. We need to enable  $inter[g, h]$  to follow all such entries, to this end we look for the closest such ancestors. We check whether  $inter[prev(g), prev(h)]$  includes an index or an *up* or *left* link. If it does, link the *prev* link to  $[prev(g), prev(h)]$ . If it does not, it means that no pattern is represented either by node  $prev(g)$  and a node on the path from the root of  $T_S$  till node  $prev(h)$  or by a node on the path from the root of  $T_{SR}$  till node  $prev(g)$  and the node  $prev(h)$ . Therefore we can step back in both pathes of the trees to  $prev(prev(g))$  and  $prev(prev(h))$ . Such a scenario can repeat, yet  $inter[prev(g), prev(h)].prev$  which is  $[prev^*(g), prev^*(h)]$ , was already computed, due to the numbering system, and the relevant information is bound to appear at that entry, so we can follow it by assigning  $prev = inter[prev(g), prev(h)].prev$ .

■

**Example.** An example of using the recursive rule, filling the *inter* table can be seen in Figure 3 for the trees depicted in Figure 2 and the following dictionary.  $P_1 = \langle 3, 9 \rangle$ ,  $P_2 = \langle 3, 5 \rangle$ ,  $P_3 = \langle 2, 9 \rangle$ ,  $P_4 = \langle 2, 7 \rangle$ ,  $P_5 = \langle 1, 2 \rangle$ ,  $P_6 = \langle 1, 5 \rangle$ ,  $P_7 = \langle 4, 1 \rangle$ ,  $P_8 = \langle 3, 4 \rangle$ ,  $P_9 = \langle 2, 3 \rangle$ ,  $P_{10} = \langle 4, 8 \rangle$ . Note the dashed *prev* arrow from  $inter[3, 8]$  to  $inter[1, 5]$ , due to the lack of information in  $inter[2, 6]$ , which is the entry of the immediate ancestors of  $[3, 8]$ .

Lemma 3 gives the preprocessing time and space guarantee.

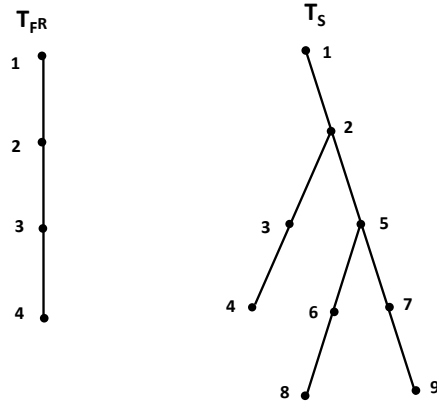
**Lemma 3.** *Preprocessing to build the inter table  $r$  requires  $O(|D| + d^2)$  time.*

*Proof.* The preprocess requires labelling both suffix trees in BFS order all in  $O(|D|)$ . Filling  $d$   $inter[g, h]$  entries with the index of the pattern the nodes  $g, h$  represent can be done in at most  $O(d^2)$ .

Filling each of the  $d^2$  entries of the table  $inter[g, h]$  can be performed in  $O(1)$  due to Lemma 2. ■

**Answering LookUp Queries.** A query of a node  $g$  from  $T_{FR}$  and node  $h$  from  $T_S$  is answered by consulting entry  $inter[g, h]$ . We output  $inter[g, h].index$  if exists which means there is a pattern whose first subpattern is represented by node  $g$  and its second subpattern is represented by node  $h$ . In order to report all relevant patterns, that their subpatterns are represented by  $g$  or its ancestors and by  $h$  or its ancestors in the suffix trees, we follow the links saved at the current entry, as detailed in the procedure appearing at Figure 4.

Lemma 4 gives the query time guarantee.



**Fig. 2.** Two suffix trees are shown, where the nodes representing subpatterns are marked by numerical labels.

$g \setminus h$	1	2	3	4	5	6	7	8	9
1	up = -- l = -- indx = 5	up = -- l = -- indx = 5	up = -- l = [1,2] indx = 9	up = -- l = [1,2] indx = 8	up = -- l = [1,2] indx = 6	up = -- l = [1,5] indx = 4	up = -- l = [1,5] indx = 4	up = -- l = [1,5] indx = 4	up = -- l = [1,5] indx = 3
2	up = -- l = -- indx = 7	up = [1,2] l = -- indx = 4	up = [1,2] l = -- indx = 9	up = -- l = -- indx = 8	up = [1,5] l = -- indx = 2	up = -- l = [3,5] indx = 4	up = [2,7] l = [3,5] indx = 4	up = -- l = [3,5] indx = 4	up = [2,9] l = [3,5] indx = 1
3	up = -- l = -- indx = 7	up = [1,2] l = [4,1] indx = 4	up = [2,3] l = [4,1] indx = 4	up = [3,4] l = [4,1] indx = 4	up = [3,5] l = [4,1] indx = 4	up = -- l = [4,1] indx = 4	up = [2,7] l = [4,1] indx = 4	up = -- l = [4,1] indx = 10	up = [3,9] l = [4,1] indx = 4

**Fig. 3.** The lookup table built according to the trees depicted in Figure 2. The arrows represent the *prev* links.

---



---

LOOKUPQUERY( $g, h$ )	
<hr/>	
1	<b>If</b> $inter[g, h].index \neq null$
2	Output $inter[g, h].index$
3	<b>If</b> $inter[g, h].prev \neq null$
4	Let $[g', h'] \leftarrow inter[g, h].prev$ .
5	LOOKUPQUERY( $g', h'$ )
6	Let $G \leftarrow g$ .
7	<b>While</b> $(inter[g, h].up \neq null)$ .
8	Let $[g', h] \leftarrow inter[g, h].up$ .
9	Output $inter[g', h].index$
10	$g \leftarrow g'$ .
11	<b>While</b> $(inter[G, h].left \neq null)$ .
12	Let $[G, h'] \leftarrow inter[G, h].left$ .
13	Output $inter[G, h'].index$
14	$h \leftarrow h'$ .

---

**Fig. 4.** The Lookup Query Procedure

**Lemma 4.** *Using the  $inter$  table, the intersection between the subpatterns appearing at location  $t_\ell$  and the reversed subpatterns ending at  $t_{\ell-gap-1}$  can be computed in time  $O(occ)$ , where  $occ$  is the number of patterns found.*

*Proof.* The query procedure is based on following links and reporting indices found. Every step of following an *up* or *left* link implies that another pattern is reported, as those links connect two subpatterns including one another, where both should be reported. The *prev* link either directs us to an entry including a pattern index, needs to be reported or it directs us to an entry with an *up* or *left* links hence, by following at most two links we encounter an index needed to be reported. Consequently, the time of following links is attributed to the size of the output. ■

As the Lookup Query procedure can replace the intersection by range queries, which is executed  $n(\beta - \alpha)$  times, Lemma 4 proves the second part of Theorem 1.

### 3.1 Splitting the Text

Usually, the input text is very long and arrives on-line. This makes the query algorithm requirement to insert all suffixes of the text unreasonable. Transforming this algorithm into an online algorithm seems a difficult problem. The main difficulty is working with on-line suffix trees construction in a sliding window. While useful constructions based on Ukkonen's [25] and McCright's [17] suffix

trees constructions exist (see [21]), no such results are known for Weiner’s suffix tree construction, which our reversed prefixes tree construction depends on.

Nevertheless, we do not need to know the whole text in advance and we can process only separate chunks of it each time. To do this we take  $m = \beta - \alpha + \max_i \sum_j |P_{i,j}|$  and split the text twice to pieces of size  $2m$ : first starting from the beginning of the text and the second starting after  $m$  symbols. We then apply the algorithms for a single gap or  $k$ -gaps for each of the pieces separately for both text splits. Note that any appearance of a dictionary pattern can still be found by the algorithms on the splitted text.

## 4 Conclusions and Open Problems

We showed that combinatorial string methods other than Aho-Corasick automaton can be applied to the *DMG* problem to yield efficient algorithms. In this paper we focused on solving *DMG*, where a single gap exists in all patterns in the dictionary. We also relaxed the problem so that all patterns in the dictionary have the same gap bounds. It is an interesting open problem to study the general problem without these relaxations.

## References

1. A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.
2. A. Amir and G. Calinescu. Alphabet independent and dictionary scaled matching. *J. of Algorithms*, 36:34–62, 2000.
3. A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.
4. A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.
5. A. Amir, D. Keselman, G. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. *Journal of Algorithms*, 37:309–325, 2000. (Preliminary version appeared in WADS 99.).
6. P. Bille, I.L. Gørtz, H. W. Vildhøj, and D. K. Wind. String matching with variable length gaps. *Theoretical Computer Science*, (443):25–34, 2012.
7. P. Bille and M. Thorup. Faster regular expression matching. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5555 of *LNCS*, pages 171–182. Springer, 2009.
8. P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1297–1308, 2010.
9. G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, pages 65–74. LNCS 1075, Springer, 1996.
10. R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don’t cares. In *Proc. 36th annual ACM Symposium on the Theory of Computing (STOC)*, pages 91–100. ACM Press, 2004.

11. T. M. Chan, K. G. Larsen, and M. Pătraşcu, *Orthogonal range searching on the ram, revisited*, Proc. 27th ACM Symposium on Computational Geometry (SoCG), 2011, pp. 1–10.
12. K. Fredriksson and S. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes and transposition invariance. *Inf. Retr.*, 11(4):338–349, 2008.
13. T. Haapasalo, P. Silvasti, S. sippu, and E. S. Soininen. Online dictionary matching with variable-length gaps. In *Proc. 10th Intl. Symp. on Experimental Algorithms (SEA)*, number 6630 in LNCS, pages 76–87. Springer, 2011.
14. K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database. *Nucleic Acids Res.*, (27):215–219, 1999.
15. M. Krishnamurthy, E. S. Seagren, R. Alder, A. W. Bayles, J. Burke, S. Carter, and E. Faskha. *How to Cheat at Securing Linux*. Syngress Publishing, Inc., Elsevier, Inc., 30 Corporate Dr., Burlington, MA 01803, 2008. e-edition: <http://www.sciencedirect.com/science/book/9781597492072>.
16. G. Kucherov, and M. Rusinowitch. Matching a set of strings with variable length dont cares. *Theoret. Comput. Sci.*, 178(12):129–154, 1997.
17. E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
18. M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo. Structured motifs search. *J. Comput. Bio.*, 12(8):1065–1082, 2005.
19. G. Myers. A four-russian algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.
20. G. Myers and G. Mehltau. A system for pattern matching applications on biosequences. *CABIOS*, 9(3):299–314, 1993.
21. J.C. Naa, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(3):87–101, 2003.
22. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.
23. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2004.
24. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth. Finding patterns with variable length gaps or don’t cares. In *Proc. 12th Annual Conference on Computing and Combinatorics (COCOON)*, pages 146–155, 2006.
25. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
26. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.
27. Verint. Packet intrusion detection. Personal communication, 2013.
28. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
29. M. Zhang, Y. Zhang, and L. Hu. A faster algorithm for matching a set of patterns with variable length don’t cares. *Inform. Process. Letters*, 110:216–220, 2010.